Kuro5hin
technology and culture, from the trenches

A Shallow Introduction to the K Programming Language (Technology)

By jjayson
Thu Nov 14th, 2002 at 05:58:07 AM EST

About two years ago I was introduced to a programming language that I really didn't like: it didn't have continuations, I didn't see any objects, it had too many operators, it didn't have a large community around it, it was strange and different, and it looked like line noise, like Perl, and I don't like Perl. However, I gave it a try.

I had to learn that continuations may not be there, but first-class functions are; it may not have a normal object system, but that is because the language doesn't need it and gets it power by cutting across objects; all the operators are the functions that make up its standard library; it's community may not be large, but it is incredibly intelligent; it only looks strange until you understand its concepts; and well, it will always look like line noise, but you will stop caring because this also make the concise code easier to read. K has since become my language of choice.

Introduction and Terminology

K is a high level system programming language available for Windows, Solaris, Linux, and maybe a few other Unix flavors upon request. It is mixture of APL, a functional programming language, and many unique features not found elsewhere. The K system is produced by Kx. They produce the language and one of the fastest commercial relational database available, KDB — written in K. You can get copies of the interactive interpreter free for educational, research, or toy purposes, and there is a non-interactive, runtime-only interpreter that is strictly free. I have heard rumors that there would be interest in opening up the language if there would be enough community response around it.

Contrary to some opinion, K is a real-world language, made for real use and sold as a real product. It is not a toy language. It is not a Turning tarpit language. It is not intentionally obfuscated or made to be intentionally strange, like Befunge or Intercal.

K was originally developed by Arthur Whitney, who was a very influential member of the APL community. Before K, Arthur was asked to write the Dr. Iverson's APL-successor, the J language, but

he decided to go his own way. This isn't his first language either; he is also the creator of the Morgan Stanley financial language A+. From 1994 to 1997 the Union Bank of Switzerland purchased exclusive rights for the use of K, but now it is available to all of us.

One of the large draws of K is the extreme programmer productivity it offers, its incredibly fast execution speeds, and its very small executable units. One of the proofs of these claims is Brian Kernighan's *Experiments with Scripting and User-Interface Languages*, in which K easily outperforms other favorite languages. Another example of these claims met is how KDB outperforms Oracle on TPC benchmarks in both query speed and data storage size while essentially being written by one person. Also, K code is very dense, and it is typical to see 100:1 code size reduction when migrating to from C to K. I have heard of almost a 1000:1 reduction when a project moved from Java and SQL to K and KSQL (KDB's query language).

K is an exceptional language for dealing with mathematical analysis, financial prediction, or anything that handles bulk data. I have used it for a document search engine and other computational linguistic tasks. K doesn't work very well when the problem is not vectorizable and scalars are the largest datum to operate on. Often though, algorithms can be written in either method. However, some problems have been almost impossible to vectorize, such as Ackermann's function, and are not good to attempt in K.

K has bindings to other popular language such as C, Java, VisualBasic, and Excel. There has also been work done on bindings to Python and Mozilla's XUL. K's builtin interprocess communication and binary format for objects is very simple and documented so making other systems interact with K is often equally simple.

Even though K is an interpreted language, the source code is somewhat compiled internally, but not into abstract machine code like Java or Python. The interpreter can be invoked interactively or non-interactively, and you can also compile a source file down to a binary file for product distribution, if you wish.

One of the hardest things for many people to get over at first is the way K looks. Even the strongest K enthusiast will freely admit that K tends to look like line noise. I came from a Scheme background and already had my eyes hurt a couple times trying to learn Perl. When my roommate introduced me to K I ridiculed it and couldn't imagine ever wanting to use it. However, after a few sessions, my eyes began to relax. Unlike Perl, the syntax is extremely regular and there is almost no syntactic sugar or special cases. K is even translatable into English; there is a program that will take a K expression and produce its English translation. To allow this translation each operator is given is a short English name and sometimes small combinations of operators are also

given common names. This is similar to building a vocabulary in a natural language. You will begin to look at larger segments of code and abstract away the actual details of what happens to the data, while understanding the higher-level concepts and transformations more easily.

To make even stronger connections with linguistics, grammatical terms are used to describe K. Operators are called verbs, and data is called nouns. There are also operators that modify other operators (these will be described later) that are called adverbs. Stringing some nouns, verbs, and adverbs together will produce clauses and sentences. This dialogue has been inherited from APL and is often abandoned for the more commonplace names of operators, functions, and variables.

Comments and Conventions

When small code segments are presented, I will use the conventions of the language and interpreter for comments and display. Forward-slash (/) is special when it comes at the beginning of a line or it has space to the left of it, then it starts a comment that lasts until the end of the line. Some people will also disguise comments as strings, since that will not effect the results of a computation. The interpreter prompts the user with two spaces to signify that it is ready to accept input. It will print all of its output without any leading space. I will do the same, showing interpreter input preceded by two spaces and interpreter output unindented.

K tends to favor simplicity over sugar. One thing that may confused people easly is K has no precedence rules. Everything is parsed from right to left. This makes it easy to see what happens next in a computation, but it also takes a little getting used to. For example, 3*2+1 in K will produce 9, instead of the more usual 7 in other language. This is always the case, except when a set of parentheses is come across. What ever inside the parentheses is first evaluated then normal evaluation resumes. People will frequently play with the order of their statements to reduce the number of characters in the expression, instead of placing parentheses around it.

Values

There are four simple types of values in K -- integer, floating point, character, and symbol; a special null type for the singular value of `_n`; and two composite types -- dictionaries and lists. The composite types are containers that others values may be stored and retrieved from. Lists are further classified as either homogenous or heterogeneous, and all homogenous lists carry the type of value they contain around with them. A list where all the simple elements are of the same type are called vectors. K is optimized to handle these homogenous lists and vectors since they appear in programming so frequently.

Lists are represented in different ways. The most general case is surrounded by parentheses with each element separated by a semi-colon. If the list is homogenous the parentheses and semi-colons will be elided and you will simply see the elements separated by a space. If all the elements are characters then the lists will be a string surrounded by double-quotes. To index a list you use brackets (`[]`) with multiple indices allowed. To index a multidimentional list use a semi-colon to separate the dimensions, but leaving a dimension blank selects all (this the same as using `_n` as the index).

A symbol is an interned string that the interpreter uses for variable lookup. They are created by the backtick followed by any legal variable name: a letter or underscore followed by any combination of letters, numbers, underscores, and dots -- but no more than two consecutive dots (since they have a special meaning that will re explained later). If you would like the symbol to not follow a legal variable name patters, you may enclose it in double quotes.

Examples:

```
   (1; 2.3; 4; 5.6)        / heterogeneous list of integers and
floats
   1 2 3 4 5               / homogeneous list of integers
   1.2 3.4 5.6             / homogeneous list of floats
   "quack"                 / homogeneous list of characters
   (1 2; 3.4 5.6; "meow") / a list of lists
   (1;2 3;4 5 6)           / a vector of integers

   "abcdefghijlkmnopqrstuvwxyz"[14 8 13 10]
"oink"

   ("qwerty";"poiuy";"asdf";"jhgfdsa")[;3] / slicing or projection
"ruff"

   `rusty
   `"http://www.kuro5hin.org"
```

Verbs

K is a very small language, so it needs to make the most of everything. Unlike APL, It uses only ASCII, so it already starts off with not too many characters to use and abuse. To allow for compact code operators are overloaded with two cases: a monadic (one argument) and diadic (two argument) use. Sometimes these uses are related and sometimes they are not. And each case the operator sometimes has slightly different behavior depending on the type or domain of the arguments.

Examples:

```
   !4       / enumerate: a list of integers from 0 to x-1
0 1 2 3

   5!3      / mod: the residue of the left modulus the right
2

   2!1 2 3 / rotate: spins right back-to-front left number of
positions.
```

```
3 1 2

  ,2  / enlist: a one item list containing only 2
,2

  1,2 / join: forms one list of the left and right argument
1 2

  |1 2 3 / reverse: reverses a list
3 2 1

  0|1    / max: the maximum also boolean OR
1

  &1 2 3 4      / where: returns the number of units specified
0 1 1 2 2 2 3 3 3 3

  &0 1 1 0 0 1 / where: an important use to get the indices of the
1s
1 2 5

  0&1           / min: the minimum also boolean AND
0

  4<5    / less-than: predicate of "is the left smaller than the
right"
1
  <7 4 9 / grade-up: sorts indices in ascending order
1 0 2

  =1 0 1 0 0 3 0 1 3 1 / group: groups all indices of same value
(0 2 7 9
 1 3 4 6
 5 8)

  2=0 1 2 3 4           / equals: compares values
0 0 1 0 0

  ?1 0 1 0 0 3 0 1 3 1  / unique: all unique elements in order seen
1 0 3

  1 0 1 0 0 3 0 1 3 1?3 / find: the first indice of the right in
left
5
```

Variables and Bindings

K is a dynamically, strongly typed language and, variables are not declared, but they come into existence when you assign a value to it. This can be done anywhere, even in the middle of an expression since there is no distinction between statements or expressions. If you try to read a value from a variable that has not yet been assigned to, you will raise an error. There are also no pointers. In true functional style, when you assign to a variable a deep copy of the value is made (K does this lazily, though). Assignment is done via the colon and it is read as "gets" or "is." As a special case, when an assignment is the last thing in an expression, null is returned (this helps prevent cluttering up the display log). You can force the return of a value from an assignment statement by using a case of the monadic colon.

Examples:

```
a:"moo"        / a gets the string "moo"
b:!10          / b gets enumerate 10 (integer list from 0 to 9)
:c:b           / c gets the value of b, but changes to b do not
```

```
effect c
0 1 2 3 4 5 6 7 8 9

  :h:(g*2),g:1+2 / h get g times 2 join g, where g gets 1 plus 2
6 3

  g
3
```

User-defined Functions

Braces (`{}`) are used to create functions; they are the equivalent of lambda in Lisp. Often they are used then the resulting function is assigned to a variable, but sometimes not. To assist in making the code compact, if a function requires three or fewer arguments, K will allow you to implicitly use `x`, `y`, and `z` as the arguments. If you need more than three arguments you must declare them all. All functions return the value of the last executed statement, even if that statement return null. To call a function you use brackets (just like a list index). If you do not supply an argument when calling a function, it will project.

Examples:

```
  pyth:{_sqrt(x*x)+y*y} / notice the two implicit arguments
  pyth[30;40]
50.0

  pyth[3 15;4 20]       / cute huh.
5 25.0

  dist:{[x1;y1;x2;y2] _sqrt((x2-x1)^2)+(y2-y1)^2}
  dist[1;1;4;5]
5.0

  :d:dist[1;1]          / project or curry the first two arguments
{[x1;y1;x2;y2] _sqrt((x2-x1)^2)+(y2-y1)^2}[1;1]

  d[7;9]
10.0

  :e:dist[1;;4]         / project on first and third argument
{[x1;y1;x2;y2] _sqrt((x2-x1)^2)+(y2-y1)^2}[1;;4]

  e[2;6]
5.0

  inc:1+
  inc 8
9
```

System Functions and Variables

After running out of punctuation Arthur made system function. Every symbol beginning with an underscore is reserved for either a system variable or system function. System functions use infix, like their less readable cousins, but like user defined functions they cannot be overloaded with monadic and dyadic cases (in the next version of K this will be changed and users will be able to define infix functions and overload them with n-adic cases).

Examples:

```
  3_draw 5          / list of 3 random numbers from 0 to 4
2 2 4

  2_draw 0          / list of 2 random real numbers from 0 to 1
0.2232866 0.9504653

  4_draw-4          / deal: list of 4 random nonrepeating numbers
from 0 to 3
2 0 1 3

  4 13_draw-52      / deal a deck of cards into four piles
(29 27 10 0 23 3 28 5 24 16 40 8 22
 51 20 36 47 18 31 26 11 44 37 38 9 13
 39 42 34 50 21 6 19 46 48 45 14 43 2
 33 49 4 25 41 30 35 7 32 17 1 12 15)

  1 3 4 5 7 9_bin 4      / binary search through list returning
index
2

  1 3 4 5 7 9_binl 2 4 6 / binary seach for a list of numbers
1 2 4

  16_vs 543212      / vector from scalar: changes base to 16
8 4 9 14 12

  5 3 2_vs 21       / also does variable change of base
3 1 1

  5 3 2_sv 3 1 1    / scalar from vector: the inverse
21

  _host`kuro5hin.org          / returns ip address as integer
-815566008

  256_vs _host`kuro5hin.org / presentation form
207 99 115 72
```

Adverbs

This is where K starts to set itself from apart from most of the common programming languages in use today. You rarely write loops in K (KDB is 100% loop-free), instead you use adverbs. An adverb modifies a function, returning another function, changing the ways it operates over its arguments and what it does with it's return values. Here is a small selection of adverbs' usages (there are other uses that are not covered here).

- Over (/) modifies a diadic function and will apply the function down a list, collection the result.
- Converge (/) modified a monadic function and will continually apply the function to the previous result until either the initial value or the result of the preceding value is returned.
- Scan (\) will apply the function down a list, collection all intermediate results (this is sometimes called trace). There is a trace analog to all usages of over.
- Each (') will apply the function down lists of the same length (equal to the valence of the function).
- Each-right (/:) will hold the left argument of the function and apply the function down the list of right arguments.

Examples:

```
  +/1 2 3 4          / plus-over (sum): is similar to 1+2+3+4
10

  +\1 2 3 4          / plus-scan: returns the intermediate values
of +/
1 3 6 10

  |/5 3 7 4 2        / max-over: compares all items like 5|3|7|4|2
7

  ,/(1 2;(3 4;5);6)  / join-over: (1 2),(3 4;5),6
(1;2;3 4;5;6)

  ,//(1 2;(3 4;5);6) / flatten: explained below
1 2 3 4 5 6

  3 4_draw'-3 -4     / draw-each: (3_draw-3),(4_draw-4)
(1 2 0
 2 0 1 3)

  2_draw/:10 100     / draw-right-each: (2_draw 10),(2_draw 100)
(7 7
 45 91)
```

Let me break down flatten for you. Join is a function that takes two values. Over modifies join and the result is a function that now takes one argument (a list) and joins all the elements of that list. Looking at the above example, when join was given a nested list, it simple stitched all the elements into a single list, removing one level of depth. However, we would like to remove all level of depth. We want join-over again to remove another level of depth, and join-over again to remove another level of depth, until the list is completely flat. This is what converge will do for us. The second application of over takes this monadic function, join-over, and continually applies it until the result no longer changes. If we didn't want to flatten the topmost list, but instead we had a list of lists to flatten, we would only want to apply flatten to each element of the top-level list. That is what flatten-each (`,//'`) would do. If we wanted to keep the top two levels of depth we would write flatten-each-each (`,//''`) and this is where things start too look like line noise.

Conditionals

Although rarely used, there are a few conditionals; most often used is the colon. It is similar to cond in Lisp: it takes pairs of arguments and an optional final argument. The first argument of each pair is tested for truth (`0` is false, all other integers are true, anything besides an integer is an error). If it is true then the result of evaluating the second of the pair is returned. If it is false then the next pair is tested. If all the pairs have been exhausted, then the final argument is evaluated and the result returned. If there is no final argument and all the conditions are false, then null is returned.

Examples:

```
  :[0;"true";"false"]
"false"
```

```
  s:{:[x>0;"+"; x<0;"-"; "0"]} / returns the sign of x or 0
  s 4
"+"

  s -3
"-3"
```

Naive Primality Test

As in *Nobody Expects the Spanish Inquisition*, I will conclude the basics with a naive primality predicate and try to explain it.

```
  isprime:{&/x!/:2_!x}  / min over x mod right-each 2 drop
enumerate x
  isprime 14
0

  isprime 7
1
```

Analyzing from right to left. We create a list of all integers from `0` to `x` (exclusive), then we remove the first two elements (`2_`), so we are left with a list from `2` to `x` (exclusive). Next we determine the residue of `x` and each of the numers in the list. Finally, we calculate the minimum of the residues. If the number is prime, then the lowest residue will be `1` and be considered true. If the number was composite there will be a 0 residue for some value.

```
  !14
0 1 2 3 4 5 6 7 8 9 10 11 12 13

  2_!14
2 3 4 5 6 7 8 9 10 11 12 13

  14!/:2_!14
0 2 2 4 2 0 6 5 4 3 2 1

  &/14!/:2_!14
0
```

The K-tree

Here is a quick, small look at one of the unique elements of K, called the K-tree. It falls nicely into the K philosophy of keeping things simple and powerful. The K-tree can be used for modularization of programs, as a scoping mechanism, for GUI design, and as a rudimentary object system.

All variables exist somewhere on the K-tree. To reference a variable you separate the name of the variable and each branch name with periods. The root of the tree is an empty symbol. For example, you might have a fully qualified variable named `.branch.subbranch.variable`.

A branch is really just a dictionary. If I were to assign a dictionary that contained symbols `sym` and `bol` to the variable `.tr.ee` then you would be able to access `.tr.ee.sym` and `.tr.ee.bol`. It goes the other way, too. If you were to create the variables `.dict.ion` and `.dict.ary` then the variable `.dict` would be a valid dictionary. This

makes the language very reflective since you can now modify variables locations, scopes, and manually manipulate extents.

Whenever you are in the K environment you are always running in a branch of the K-tree. When the K interpreter starts it places you in the `.k` branch. You can move around the tree with the directory (`\d`) command. For example `\d .tw.ig` would create the new branch `tw` then create a subbranch `ig`. You can inspect all the variables in a branch with the list variables (`\v`) command. Often a script will begin with a change directory command and then define all of its variables in that (and maybe a few more) branches. This effectively uses the K-tree as a module system.

```
   \d .test           / create a new directory off the root
   \d ement           / create a sub-branch
   \d                 / show the current directory
.test.ement

   new:`small         / put some values in the directory
   old:`big
   \v                 / list the contents o
new old

   \d ^               / back up one directory in the tree
   \d
.test

   \v
ement

   ement              / inspect the value of ement
.((`new;`small;)
   (`old;`big;))

   .test.ement.new    / fully-qualified
`small

   ement.old          / partially qualified
`big

   ement`old          / index like an array
`big
```

Other Unique Elements of K

The K philosophy itself has some unique insights on how to treat data and the appropriate way to organize data for efficient processing, however within the language there are some features that you will not find in any other language:

K uses the tree to hold an attribute structure. These attributes are used for documentation strings, GUI representations, for other functionality in K, and for whatever you decide to use them for. Some of the programming environments for K make entensive use of attributes to store information about where functions are defined and other administrative information.

K also has a the concept of dependencies and triggers. They are used for efficient, demand-based calculation of data and executing callback code whenever a value is changes (this is how the timer system in K works). K will keep track of out of date dependencies

for you and only do the minimal amount of work necessary to update values.

K has a unique, declarative GUI subsystem that is based around the K tree and triggers. It takes a minimal amount of work to put a GUI on your application. K takes the approach that you show variables and changes made to that variable on the screen are immediately reflected in the program without any work by you.

K's interprocess communication (IPC) and network communication systems is also based around callbacks and message handlers. There are simple K primitives that will ship entire K data structures around for you, or you may do it yourself. The goal of the IPC system is like the goal with rest of K, make it fast, simple, powerful, and highly useful.

In a time when programming languages are lacking in originality and are not bringing new ideas to the table, K succeeds where others fail. But K is not just a research language, not appropriate for real-world use. While the community may be small some of the users of the language are very big. With recent implementations by Island ECN and the US government, this looks to only be getting bigger, too. The next version of the language will fix many of the nagging holes and annoyances and take away the line noise factor that has pushed many away.

Here are some closing simple segments of K that are informative on how the K way of approaching a problem may be different:

```
   cut:{1_'(&x=*x)_ x:" ",x}
   cut "Scoop ate my spaces"
("Scoop"
 "ate"
 "my"
 "spaces")

   fibonacci:{x(|+\)\1 1}
   fibonacci 5
(1 1
 2 1
 3 2
 5 3
 8 5
 13 8)

   first:*:
   euclid:{first(|{y!x}\)\x,y} / Euclid's Algorithm
   euclid[24;40]
(24 40
 16 24
 8 16
 0 8)
```

Full discussion:
http://www.kuro5hin.org/story/2002/11/14/22741/791